UNIVERSITETET
I OSLO

Control in Sequential Languages

Arild B. Torjusen

aribraat@ifi.uio.no

Department of Informatics – University of Oslo

Based on John C. Mitchell's slides (Stanford U.) , adapted by Gerardo Schneider, UiO.

INF 3110/4110 - 2006

ML lectures

1. 04.09: The Algol Family and ML (Mitchell's chap. 5 + more)
2. 11.09: More on ML & types (chap. 5 and 6)
3. 18.09: More on Types, Type Inference and Polymorphism (chap. 6)
4. **25.09: Control in sequential languages, Exceptions and Continuations (chap. 8)**

Outline

- Structured Programming
  - *go to* considered harmful
- Exceptions
  - “Structured” jumps that may return a value
  - Dynamic scoping of exception handler

Control flow in sequential programs

- The execution of a (sequential) program is done by following a certain control flow
- The end-of-line (or semi-colon) terminates a statement
- What is the next instruction to be executed?
  - The flow of control goes top-down in general
  - Jumps (loops, conditionals, etc)
- It is not easy, in general to ”see” whether a given instruction is reachable from another (Program Analysis)

Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30  X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   ...
```

Just a label

Similar structure may occur in assembly code

Historical Debate

- Dijkstra: "Go To Statement Considered Harmful" (1968)
  - "... the **go to** statement should be abolished from all 'higher level' programming languages..."
- Knuth: "Structured Programming with go to Statements" (1974)
  - You can use goto, but do so in structured way ...
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Advance in Computer Science

- Standard constructs that structure jumps

  if ... then ... else ... end

  while ... do ... end

  for ... { ... }

  case ...

- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body
  - Exceptions and continuations (?!)

Jumps into Blocks – Why not?

- Label in the body of a function
- Should an activation record be created?
- If not, what about local variables?
  - They are meaningless
- If so, how to set function parameters?
  - There are no parameter values

```
fun bizarre(pars);
  local vars;
   ...
  a: ....
   ...
end;

Program P;
  ....
  goto a;
  ....
end;
```

No clear answers! Better to reject these programs!

Outline

- Structured Programming
  - *go to* considered harmful

- **Exceptions**
  - “Structured” jumps that may return a value
  - Dynamic scoping of exception handler

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
- Memory management needed
  - Unnecessary activation records may be deallocated
- Two main language constructs
  - Statement or expression to *raise* or *throw* exception
  - Declaration to establish exception *handler*
- Possible to have more than one handler

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant;  (* declare exception name *)
fun invert (M) =        (* function to invert matrix *)
        ...
        if Det = 0
            then raise Determinant    (* exit if Det=0 *)
            else ...
 end;
...
invert (myMatrix) handle Determinant => ... ;
```

Value for expression if determinant of myMatrix is 0

ML Exceptions

- Exceptions are a different kind of entity than types
- Declare exceptions before use
- Exceptions are dynamically scoped
  - Control jumps to the handler most recently established (run-time stack) (more later…)
  - ML is otherwise statically scoped.
- Pattern matching is used to determine the appropriate handler (C++/Java uses type matching)

ML Exceptions

- Declaration

  exception ⟨name⟩ of ⟨type⟩

  gives name of exception and type of data passed when raised

- Raise

  raise ⟨name⟩ ⟨parameters⟩

  expression form to raise and exception and pass data

- Handler

  ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩

  evaluate first expression exp1

  if exception that matches pattern is raised,

  then evaluate second expression exp2 instead

General form allows multiple patterns.

ML Exceptions - example

```
- exception noSuchElement ;
- fun nth (n,nil) = raise noSuchElement
    | nth (0,s::ss) = s
    | nth (n,s::ss) = nth((n-1),ss) ;
 val nth = fn : int * 'a list -> 'a
 - nth(2,[1,2,3]) ;
 val it = 3 : int
 - nth(4,[1,2,3]) ;
 uncaught exception noSuchElement
   raised at: stdIn:10.25-10.38
 - fun safeNth(n,xs) = nth(n,xs) handle noSuchElement => 0 ;
 val safeNth = fn : int * int list -> int
 - safeNth(4,[1,2,3]) ;
 val it = 0 : int
```

Which Handler is Used?

```
exception Ovflw;
fun reciprocal(x) =
      if x<=min  then raise Ovflw  else 1.0/x;
(reciprocal(x) handle Ovflw=>0.0)  /  (reciprocal(x) handle Ovflw=>1.0);
```

- Dynamic scoping of handlers
  - First call handles exception one way
  - Second call handles exception another
  - General dynamic scoping rule

    Jump to most recently established handler on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not

Handlers with pattern matching

```
- exception Signal of int;
- fun f(x) = if x=0   then raise Signal(0)
         else if x=1   then raise Signal(1)
         else if x=10 then raise Signal(x-8)
         else (x-2) mod 4;
```

```
- f(10) handle Signal(0) => 0
             | Signal(1) => 1
             | Signal(x) => x+8;
> val it = 10 : int
```

- The expression to the left of the handler is evaluated
- If it terminates normally the handler is not invoked
- If the handler is invoked, pattern matching works as usual in ML

Exception for Error Condition

```
- datatype ‘a tree = LF of  ‘a | ND of  (‘a tree)*(‘a tree);
- exception No_Subtree;
- fun lsub (LF x) = raise No_Subtree
  |    lsub (ND(x,y)) = x;
> val lsub = fn : ‘a tree -> ‘a tree
```

- This function raises an exception when there is no reasonable value to return

```
- lsub(LF(3));
> uncaught exception No_Subtree raised at:...

- lsub(ND (LF(3),LF(5)));
> val it = LF 3 : int tree
```

INF 3110/4110 - 2006

Exception for Efficiency

- Function to multiply values of tree leaves

```
- fun prod(LF x) = x: int
  |    prod(ND(x,y)) = prod(x) * prod(y);
```

- Optimize using exception

```
- fun prod(tree) =
      let exception Zero
          fun p(LF x) = if x=0 then (raise Zero) else x
          |    p(ND(x,y)) = p(x) * p(y)
      in
          p(tree) handle Zero => 0
      end;
```

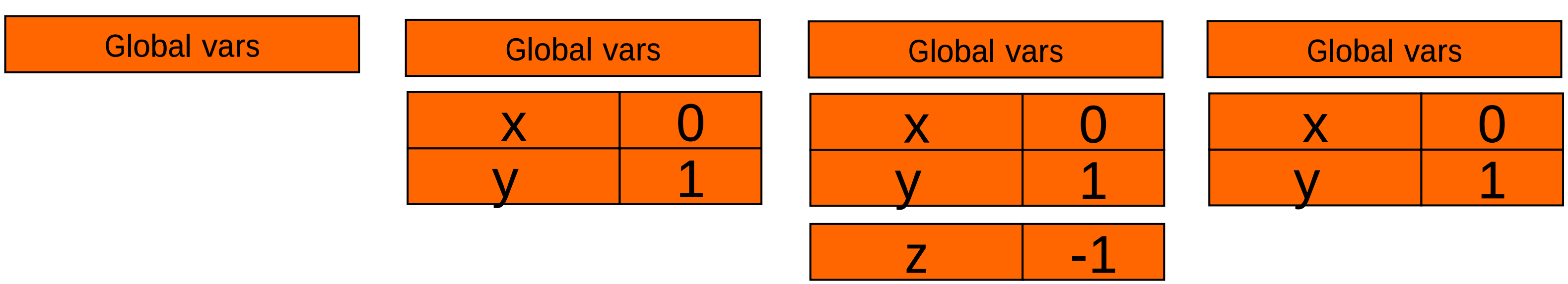
Runtime organization, a preview

- Block structured languages, in-line blocks
  - C/C++/Java {…}
  - in ML each declaration is a separate block
  - When a program enters a new block an activation record is added to the run-time stack

Runtime organization, a preview

- Run-time stack

```
…
{int x=0;
 int y = x + 1;
        {int z = (x+y) * (x-y);}
};
…
```



Runtime organization, a preview

- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables
  - Access link (aka static link)
  - Control link (dynamic link)

Dynamic Scope of Handler

```
- exception X;
(let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
 in
    g(f) handle X => 4
 end) handle X => 6;
```

scope

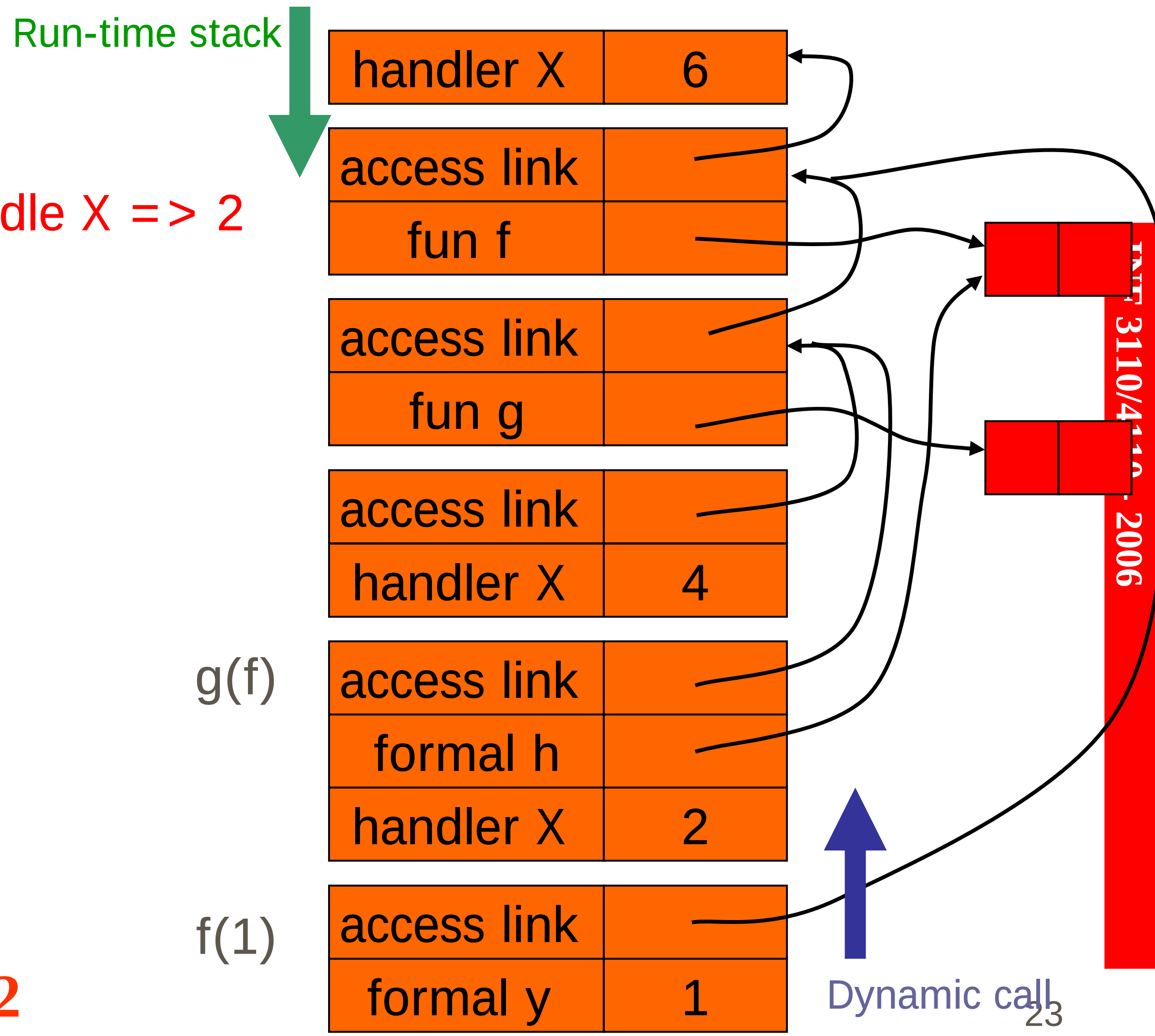
handler

What is the value of g(f)?

It depends on which handler is used!

Dynamic Scope of Handler

```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```



Dynamic scope:
find first X handler, going up the dynamic call chain leading to raise X.

Answer: g(f) = **2**

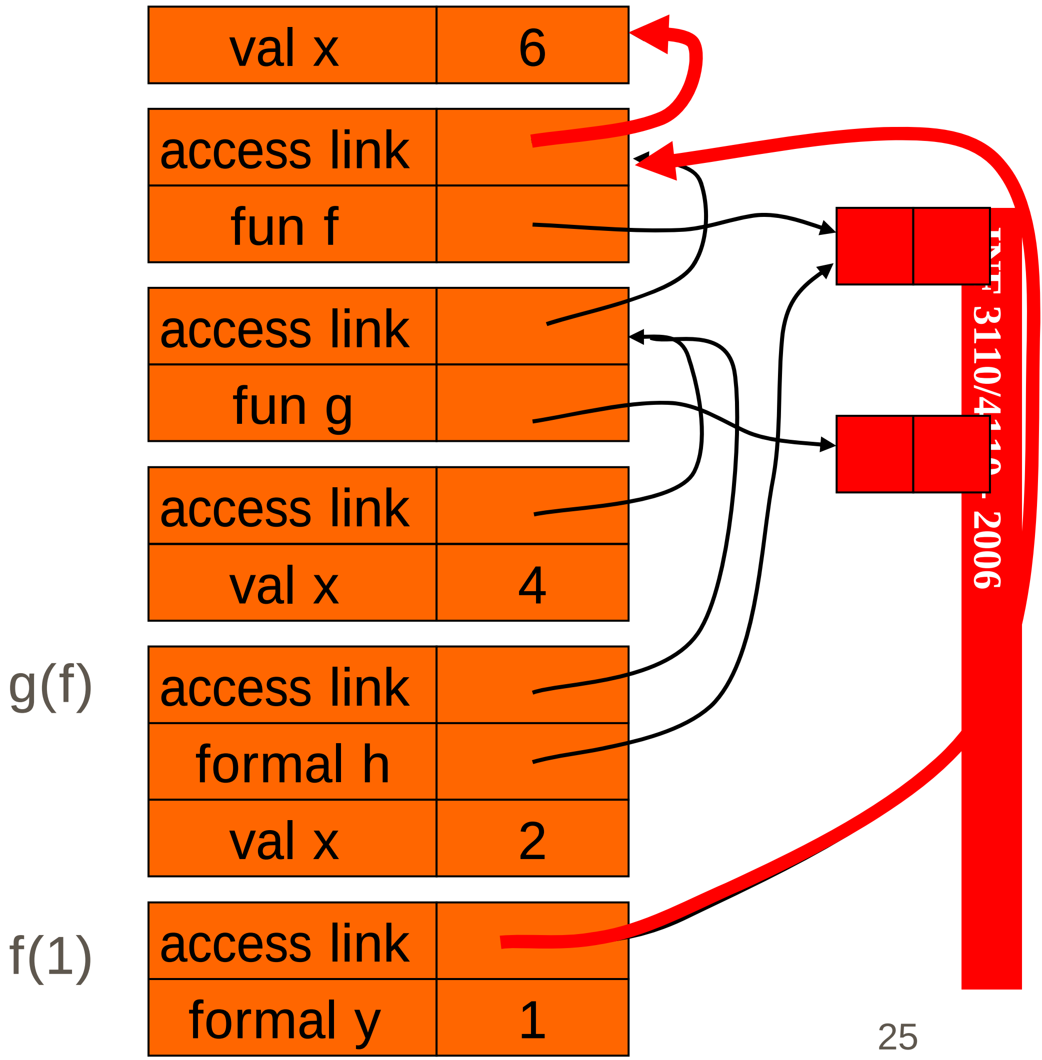
Compare to Static Scope of Variables

```
exception X;
(let fun f(y) = raise X
     and g(h) = h(1)
                    handle X => 2
in
     g(f) handle X => 4
end) handle X => 6;
```

```
val x=6;
(let fun f(y) = x
     and g(h) =
          let val x=2 in h(1)
 in
     let val x=4 in g(f)
end);
```

Static Scope of Declarations

```
val x=6;
(let fun f(y) = x
    and g(h) =
            let val x=2 in h(1)
  in
    let val x=4 in g(f)
end);
```



Static scope: find first x, following access links from the reference to x.

Answer: g(f) = **6**

Typing of Exceptions

- Typing of raise ⟨exp⟩
  - Recall definition of typing
    - – Expression e has type t if normal termination of e produces value of type t
  - Raising exception is not normal termination
    - \- 1 + raise No_value (the sum will not be performed)
- Typing of handle ⟨exp⟩ => ⟨value⟩
  - Converts exception to normal termination
  - Need type agreement
  - Examples
    - \- 1 + ((raise X) handle X => e) Type of e must be int
    - \- 1 + (e_1 handle X => e_2) Type of e_1, e_2 must be int

INF 3110/4110 - 2006

Exceptions and Resource Allocation

```
exception X;
(let
   val x = ref [1,2,3]
 in
   let
      val y = ref [4,5,6]
   in
      ... raise X
    end
end); handle X => ...
```

[1,2,3] built in the heap, ref x pushed into stack

[4,5,6] built in the heap, ref y pushed into stack

Control is transferred outside the scope

x and y popped off the stack

[1,2,3] and [4,5,6] garbage collected

Exceptions and Resource Allocation

```
exception X;
(let
   val x = ref [1,2,3]
 in
   let
       val y = ref [4,5,6]
   in
      ... raise X
    end
end);  handle X => ...
```

- Resources allocated between handler and raise may be "garbage" after exception
- Open files might not be closed

General problem: no obvious solution

Further Reading

- Mitchell's chapter 8

ML summary

- Is ML unpractical?, what about
  - Input/Output, using files
  - Interacting with underlying OS
  - Making executable applications
  - etc. etc.
- We have focused on the basics
  - Basic ML constructs
  - Learning to think ”functional”, recursion
  - Higher order functions
  - Type system and type inference
  - Exceptions

UNIVERSITETET
I OSLO

Something on non-Java-like languages

Arild B. Torjusen
aribraat@ifi.uio.no

Department of Informatics – University of Oslo

INF 3110/4110 - 2006

- **Based on slides by Gerardo Schneider, UiO.**

Outline

- Why (not only) Java?
- Some successful stories of non-Java-like languages

INF 3110/4110 - 2006

Object-oriented programming: a small part of a big world

- Object-oriented programming is just one tool in a vastly bigger world
- For example, consider the task of building robust telecommunications systems
  - Ericsson has developed a highly available ATM switch, the AXD 301, using a message-passing architecture (more than one million lines of Erlang code)
  - The important concepts are isolation, concurrency, and higher-order programming
  - Not used are inheritance, classes and methods, UML diagrams, and monitors

Source: Peter Van Roy's slide of an invited talk at CLEI'05 – Cali, Colombia

Something on concurrent prog.

- There are three main paradigms of concurrent programming
  - Declarative (dataflow; deterministic) concurrency
  - Message-passing concurrency (active entities that send asynchronous messages; Actor model, Erlang style)
  - Shared-state concurrency (active entities that share common data using locks and monitors; Java style)
- Declarative concurrency is very useful, yet is little known
  - No race conditions; allows declarative reasoning techniques
  - Large parts of programs can be written with it
- Shared-state concurrency is the most complicated (the worst to program in), yet it is the most widespread (e.g. Java)!
  - Multiple threads accessing shared variables
  - Interleaving semantics: huge number of cases and complicated reasoning

Source: Based on Peter Van Roy's slides of an invited talk at CLEI'05 – Cali, Colombia

Some problems with Java

- Java is based on the shared-state concurrency model
- Shared-state and message-passing models are equaly expressive (theoretically) but not in practice: The shared-state model is harder to program than the message-passing model!

Better model: objects communicating asynchronously through message-passing

Limitations of Java

- Not good for
  - Internet programming (e.g. web services)
  - Safety critical systems (e.g. aeoronautics)
  - Dynamic upgrading (e.g. telecommunication)
  - Real-time systems (e.g. robotics)
  - Embedded safe systems (e.g. Smart cards)
  - Component-based applications
  - Systems involving cross-cutting concerns like history information and synchronization (problems with the aggregation and inheritance mechanism)
  - …

Why to study ML then?

- There are enough applications of ML-like languages!
- Anyway, the intention of the course is to teach new concepts, not a new language
- ML is a simple function-oriented language with many interesting features
  - Type inference algorithm
  - Polymorphism
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions

Outline

- Why (not only) Java?
- Some successful stories of non-Java-like languages

Text editing: Lisp

- Emacs is an extensible, customizable, self-documenting real-time display editor
- It is a text editor and more. At its core is an interpreter for Emacs Lisp (”elisp'', for short), a dialect of the Lisp PL with extensions to support text editing
- Lisp is a family of functional languages. The two major dialects in use today are Common Lisp and Scheme (see http://www.lisp.org/table/lisp.htm)
- See http://www.lisp.org/table/good.htm for more applications of Lisp

Source:http://www.gnu.org/software/emacs/emacs.html#Whatis

Telecommunications: Erlang

- Erlang is a functional language for reliable concurrent and distributed systems developped at Ericsson and SICS (Sweden)
- Some applications: used by Ericsson in the phone switches AXD301, DWOS, A910, and ANx
- Highly reliable: less than 3 minutes of downtime in one year of operation

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Finance: Haskell

- Haskell is a lazy pure functional language (see http://www.haskell.org)
- Application: a combinator library for describing compositional financial contracts (by S. Peyton Jones, J.-M. Eber and J. Seward)
- A new enterprise was created based on these ideas (Lexifi Technologies)
- There are many other applications! (see http://www.haskell.org/practice.html)

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

More applications of Haskell

- Galois is a Sw development company based in U.S.A developing Sw under contract, and every project uses Haskell (also SML-NJ and OCaml)
- Some applications:
  - Cryptol, a domain-specific language for cryptography (with an interpreter and a compiler);
  - A cross-domain file and web server;
  - A GUI debugger for a specialized chip;
  - A tool for easily embedding new syntax in the client's own language;
  - A legacy code translator (translating from K&R C to ANSI C, while moving from SunOS 4 to Solaris *and* a new abstract API)

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Theorem prover 1: ML

- Isabelle is a generic theorem prover. New logics are introduced by specifying their syntax and rules of inference. Proof procedures can be expressed using tactics and tacticals
- It was developped by L.C. Paulson and T. Nipkow
- Written in SML

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Theorem prover 2: Lisp and ML

- The HOL system is a powerful computer program for constructing formal specifications and proofs in higher order logic
- Used in both industry and academia to support formal reasoning in Hw design and verification, reasoning about security, proofs about real-time systems, semantics of Hw description lang., compiler verification, program correctness, modelling concurrency, and program refinement
- HOL88 is the original Cambridge HOL system built using Lisp; HOL90 is a reimplementation in SML

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Theorem prover 3: ML

- ALF is an interactive theorem prover (a proof editor) based on Martin-Löf's type theory with explicit substitution
- The proof engine of ALF is written in SML
- Developers: Thierry Coquand, Lena Magnusson and Bengt Nordström/ Programming Logic Group at Chalmers, Sweden

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Verification tool: ML and Lisp

- The Java PathFinder, JPF, is a translator from Java to Promela, the programming language of the SPIN model checker. The purpose is to establish a framework for verification and debugging of Java programs using model checking. The system detects deadlocks and violations of assertions stated by the programmer
- Written in Common Lisp and Moscow ML
- Developed at NASA Ames (USA)

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Web, HTML, XML

- SXML and SXML tools are S-expression-based implementations of W3C XML recommendations, the embedding of XML data and XML query and manipulation tools. Written in Scheme
- HaXml is a collection of utilities for parsing, filtering, transforming, and generating XML documents using Haskell.

Source: http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

More applications of non-Java like PL

- Much, much more: I haven't mentioned here much on actor- and aspect-oriented programming, Internet programming, real-time languages, embedded systems, logic programming (Prolog), etc
- For more applications of functional languages see for instance "Functional Programming in the Real World":

http://homepages.inf.ed.ac.uk/wadler/realworld/index.html